

Modelling Delegation and Revocation Schemes in IDP

Marcos Cramer, Pieter Van Hertum, Diego Agustín Ambrossio, Marc Denecker

University of Luxembourg
{marcos.cramer,diego.ambrossio}@uni.lu
KU Leuven, Belgium
{pieter.vanhertum,marc.denecker}@cs.kuleuven.be

Abstract. In ownership-based access control frameworks with the possibility of delegating permissions and administrative rights, chains of delegated accesses will form. There are different ways to treat these delegation chains when revoking rights, which give rise to different revocation schemes. In this paper, we show how IDP – a knowledge base system that integrates technology from ASP, SAT and CP – can be used to efficiently implement executable revocation schemes for an ownership-based access control system based on a declarative specification of their properties.

Keywords: access control, delegation, revocation, IDP, logic programming, knowledge base system

1 Introduction

In ownership-based frameworks for access control, it is common to allow principals (users or processes) to grant both permissions and administrative rights to other principals in the system. Often it is desirable to grant a principal the right to further grant permissions and administrative rights to other principals. This may lead to delegation chains starting at a *source of authority* (for example the owner of a resource) and passing on certain permissions to other principals in the chain.

Furthermore, such frameworks commonly allow a principal to revoke a permission that she granted to another principal. Depending on the reasons for the revocation, different ways to treat the chain of principals whose permissions depended on the second principal's delegation rights can be desirable. For example, if one is revoking a permission given to an employee because he is moving to another position in the company, it makes sense to keep in place the permissions of principals who received their permissions from this employee; but if one is revoking a permission from a user who has abused his rights and is hence distrusted by the user who granted the permission, it makes sense to delete the permissions of principals who received their permission from this user. Any algorithm that determines which permissions to keep intact in which permissions to delete in the case of the revocation of a permission is called a *revocation scheme*.

Hagström et al. [18] have presented a framework for classifying possible revocation schemes along three different dimensions: the extent of the revocation to other grantees (propagation), the effect on other grants to the same grantee (dominance), and the permanence of the negation of rights (resilience). Since there are two options along each

dimension, there are in total eight different revocation schemes in Hagström’s framework. This classification was based on revocation schemes that had been implemented in database management systems [17,15,6,5].

IDP is a *Knowledge Base System*, which combines a declarative specification in $\text{FO}(\cdot)$, with imperative management of the specification via the Lua scripting language. An $\text{FO}(\cdot)$ specification theory consists of formulas in first-order logic and *inductive definitions*. Inductive definitions are essentially logic programs in which clause bodies can contain arbitrary first-order formulas. The combination of the declarative specification and the imperative programming environment makes this logic programming tool suitable for solving a large variety of different problems.

In this paper, we show that revocation schemes can be efficiently implemented in IDP by modelling them as IDP theories. One of the key features that make IDP a very efficient tool for implementing revocation schemes is the possibility to use inductive definitions for defining functions and predicates in $\text{FO}(\cdot)$, since the formal definition of revocation schemes can be captured in an elegant way as an inductive definition.

The paper is structured as follows: We introduce Hagström et al.’s classification of revocation schemes in section 2. After introducing $\text{FO}(\cdot)$ and IDP in section 3, we show how we implemented the revocation schemes of Hagström et al.’s classification in IDP in section 4. Section 5 discusses related work and section 6 concludes the paper.

2 The revocation classification framework

In this section we give both a formal and an informal presentation of Hagström et al.’s [18] classification of revocation schemes.

Let P be the set of principals (users or processes) in the system, let O be the set of objects for which authorizations can be stated and let A be the set of access types, i.e. of actions that principals may perform on objects. For every object $o \in O$, there is a *source of authority* (SOA), for example the owner of file o , which is a principal that has full power over object o and is the ultimate authority with respect to accesses to object o . For any $a \in A$ and $o \in O$, the SOA of o can grant the right to access a on object o to other principals in the system, and can also delegate the right to grant access and to grant this delegation right. Additionally, our framework allows for negative authorizations, which can be used to temporarily block a principal’s access or delegation rights concerning a certain object and access type.

We assume that all authorizations in the system are stored in an authorization specification, and that every authorization is of the form (i, j, a, o, b_1, b_2) , where $i, j \in P$, $a \in A$, $o \in O$ and b_1 and b_2 are booleans. The meaning of this authorization is that principal i is granting some permission concerning access type a on object o to principal j . If b_1 is \top , then the permission is a positive permission, else it is a negative permission. If b_2 is \top , the permission contains the right to delegate the permission further. Since it does not make sense to combine a negative permission with the right to delegate the permission, the combination \perp, \top for b_1, b_2 is disallowed.

There is no interaction between the rights of principals concerning different access-object pairs (a, o) . For this reason, we can consider a and o to be fixed for the rest of the paper, and can simplify (i, j, a, o, b_1, b_2) to (i, j, b_1, b_2) .

2.1 Delegation chains and connectivity property

We first present the part of the system that does not involve negative authorizations. In section 2.2 we will introduce negative authorizations.

The right of a principal i to delegate the access right to other principals can be defined by the existence of a *rooted delegation chain*, i.e. a delegation chain connecting the SOA with i :

Definition 1. A rooted delegation chain for principal i is a chain (p_1, \dots, p_n) of principals satisfying the following properties:

1. p_1 is the source of authority.
2. p_n is i .
3. For every integer k with $1 \leq k < n$, the authorization $(p_k, p_{k+1}, \top, \top)$ is in place.

A principal j has the access right if she has delegation right or if some principal with delegation right has granted her the access right, i.e. if there is a principal i such that the authorization (i, j, \top, \perp) is in place and there is a rooted delegation chain for i .

The framework allows an authorization (i, j, b_1, b_2) to be in the authorization specification only if i has the delegation right. This is called the *connectivity property*:

Connectivity property: For every authorization (i, j, b_1, b_2) in the authorization specification, there is a rooted delegation chain for i .

We visualize an authorization specification by a labelled directed graph as in the following example:

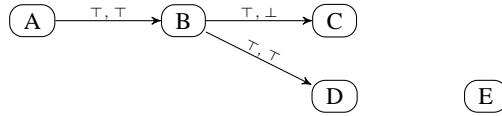


Fig. 1. Authorization specification visualized as labelled directed graph

In this example, in which A is the SOA (as in all forthcoming examples), the principals A , B and D have the delegation right, C has the access right but not the delegation right, and E has no rights concerning the access type and object in question.

2.2 Negative authorizations

A negative authorization from i to j can inactivate a positive authorization from i to j without deleting it. The purpose of this is to make it possible to temporarily take away rights from a user without deleting anything from the authorization specification, so that it is easier to go back to the state that was in place before this temporary removal of rights.

Hagström et al. [18] leave it open whether negative permissions dominate positive ones or the other way round. In this paper, we work under the assumption that positive

permission dominate negative permissions. More precisely, this means that a negative authorization (i, j, \perp, \perp) directly inactivates only positive authorizations from i to j , and leaves other permission to j active. But the connectivity property is assumed to also hold for the subset of the authorization specification that consists only of the active authorizations. Hence additionally to the directly inactivated authorizations, there may also be indirectly inactivated authorizations, as the authorization from B to C in the following example:

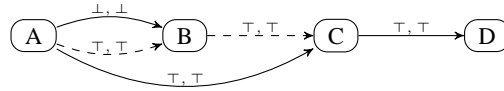


Fig. 2. The effect of a negative authorization

In this example, principal A has issued a negative authorization to principal B , thus inactivating the access and delegation rights of B . Since B no longer has the right to delegate, the authorization from B to C , which could only be issued because of B 's right to delegate, is also inactivated. But C still has a rooted delegation chain that is independent of B , so that the authorization from B to C is not affected.

In order to formally specify which authorizations get inactivated in this way, we first need to define the notion of an *active rooted delegation chain*:

Definition 2. An active rooted delegation chain for principal i is a chain (p_1, \dots, p_n) of principals satisfying the following properties:

1. p_1 is the source of authority.
2. $p_n = i$.
3. For every integer k with $1 \leq k \leq n$, the authorization $(p_k, p_{k+1}, \top, \top)$ is in place and the authorization $(p_k, p_{k+1}, \perp, \perp)$ is not in place.

Now a positive authorization (i, j, \top, b) is considered inactive if there is no active rooted delegation chain for i .

When the authorization specification contains negative authorizations, the delegation right and access right definitions of section 2.1 can no longer be applied as stated before, but must be modified by adding the word “active” to “rooted delegation chain”: A principal i has delegation right if there is an *active* rooted delegation chain for i , and a principal j has access right if there is a principal i such that the authorization (i, j, \top, \perp) is in place and there is an *active* rooted delegation chain for i .

2.3 The three dimensions

Hagström et al. [18] have introduced three dimensions according to which revocation schemes can be classified. These are called *propagation*, *dominance* and *resilience*:

Propagation. The decision of a principal i to revoke an authorization previously granted to a principal j may either be intended to affect only the direct recipient j or to affect all the other users in turn authorized by j . In the first case, we say that the revocation is *local*, in the second case that it is *global*.

Dominance. This dimension deals with the case when a principal losing a permission in a revocation still has permissions from other grantors. If these other grantors' are dependent on the revoker, she can dominate these grantors and revoke the permissions from them. This is called a *strong* revocation. The revoker can also choose to make a *weak* revocation, where permissions from other grantors to a principal losing a permission are kept.

In order to formalize this dimension, we need to define what we mean by a principal's delegation rights to be independent of another principal:

Definition 3. A principal j has delegation rights independent of a principal i iff there is an active rooted delegation chain (p_1, \dots, p_n) such that p_1 is the SOA, $p_n = j$ and $p_k \neq i$ for every $1 \leq k \leq n$.

Resilience. This dimension distinguishes revocation by removal of positive authorizations from revocation by negative authorizations which just inactivate positive authorizations. We call revocations of the first kind *deletes* and revocations of the second kind *negatives*.

2.4 The eight revocation schemes

For brevity, we just present five of the eight revocation schemes in detail, each with an example in which the authorization from A to B in the following authorisation specification is revoked according to the revocation scheme under consideration:

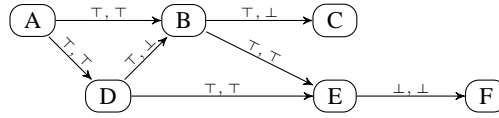


Fig. 3. Example authorization specification before revocation

Weak local delete. A *weak local delete* of a positive authorization from i to j has the following effect:

1. The authorization from i to j is deleted.
2. If step 1 causes j to lose its delegation right, all authorizations emerging from j are deleted.
3. For every authorization (j, k, b_1, b_2) deleted in step 2, an authorization of the form (i, k, b_1, b_2) is issued.

Step 2 ensures that the connectivity property is satisfied at j . This being a local revocation scheme, step 3 ensures that all rights that users other than j had before the operation are intact.

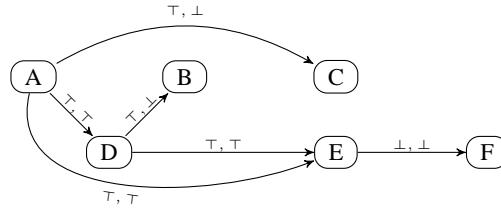


Fig. 4. Weak Local Delete from A to B

Weak global delete. A *weak global delete* of a positive authorization from i to j has the following effect:

1. The authorization from i to j is deleted.
2. Recursively, any authorization emerging from a principal who loses her delegation right in step 1 or step 2 is deleted.

The recursive step 2 ensures that the connectivity property is satisfied for the whole authorization specification after this operation.

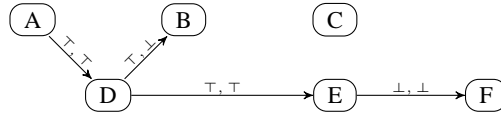


Fig. 5. Weak Global Delete from A to B

Strong local delete. A *strong local delete* of a positive authorization from i to j has the following effect:

1. The authorization from i to j is deleted.
2. Every authorization of the form (k, j, \top, b) such that k is not independent of i is deleted.
3. If steps 1 and 2 cause j to lose its delegation right, all authorizations emerging from j are deleted.
4. For every authorization (j, k, b_1, b_2) deleted in step 3, an authorization of the form (i, k, b_1, b_2) is issued.

The only difference to the weak local delete is step 2, which is the step that makes this a strong revocation scheme.

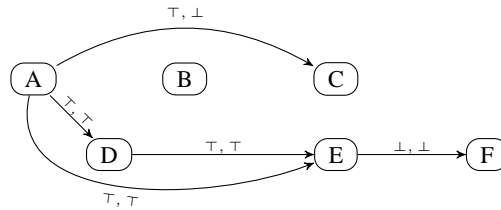


Fig. 6. Strong Local Delete from A to B

Strong global delete. A *strong global delete* of a positive authorization from i to j has the following effect:

1. The authorization from i to j is deleted.
2. Recursively, delete authorizations as follows:
 - (a) Any authorization emerging from a principal who loses her delegation right in step 1, step 2.(a) or step 2.(b) is deleted.
 - (b) Any authorization of the form (k, l, \top, b) , where l is a principal who loses her delegation right in step 1, step 2.(a) or step 2.(b) and k is not independent of i , is deleted.

Here the recursive deletion procedure contains two different kinds of deletions: 2.(a) makes it a global revocation scheme and 2.(b) makes it a strong revocation scheme.



Fig. 7. Strong Global Delete from A to B

Negative revocations. The negative revocations are similar to the positive revocations, only that instead of deleting positive authorizations, we inactivate them by issuing negative authorizations. We show this on the example of the weak global negative. The other three negative revocation schemes are adapted versions of the corresponding deletes in a similar way.

A *weak local negative* of a positive authorization from i to j has the following effect:

1. The negative authorization (i, j, \perp, \perp) is added to the authorization specification.
2. For every authorization (j, k, b_1, b_2) inactivated by step 1, an authorization of the form (i, k, b_1, b_2) is issued.

Unlike in the weak local delete, we do not delete any authorizations. The definition of *inactive* authorizations from section 2.2 ensures that authorizations that get deleted in a weak local delete get inactivated in a weak local negative, even though we do not need to state explicitly which authorizations get inactivated.

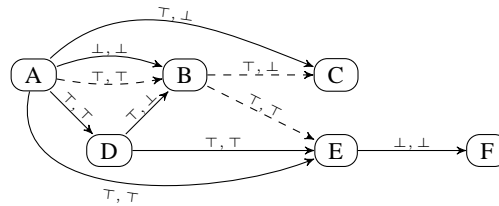


Fig. 8. Weak Local Negative from A to B

2.5 Undoing negative revocation schemes

The purpose of negative authorizations is to temporarily block someone's rights. There is an *undo operation*, which undoes the effect of such a temporary blocking. In the case of the weak global negative, the undo operation just consists of deleting the negative authorization (i, j, \perp, \perp) that was added to the authorization specification. In the case of the other three negative revocation schemes, we also need to delete the auxiliary authorizations that were issued as part of the revocation scheme. This can be achieved by labelling their dependence on the negative authorization (i, j, \perp, \perp) that was added in step 1 of the negative revocation scheme, and deleting them as soon as this negative revocation is deleted.

3 FO(\cdot) and the IDP-system

In developing the FO(\cdot)-IDP-project, the ambition is to create a full integration of existing *pure declarative KR-languages* with a *knowledge base system* to implement reasoning for these languages.

3.1 The KR-languages: FO(\cdot)

We use the term FO(\cdot) for a family of extensions of first-order logic (FO). FO(\cdot) is developed with the purpose to combine ideas from multiple domains of knowledge representation, logic programming and on monotonic reasoning in a conceptual clear manner. The basis of this family of languages lies in first order (classical) logic, extended with new language constructs from the fields of logic programming, constraint programming and non monotonic reasoning.

The IDP-system supports an FO(\cdot) language, denoted by FO(\cdot)^{IDP}. In the context of this paper, the focus lies on FO(\cdot)^{IDP}, since this is the language used for the modelling of the revocation schemes further on. This section provides an overview of the core of the FO(\cdot)^{IDP} language, more details about the language can be found in [10]. We assume familiarity with basic concepts of classical logic and logic programming. The most important extensions in FO(\cdot)^{IDP} are *types*, *arithmetic*, *(partial) functions*, *aggregates* and *inductive definitions*.

An FO(\cdot) specification consists of vocabularies, theories, terms and structures. A vocabulary Σ consists of a set of types and a set of predicate- and function symbols, annotated with the types of their arguments. A structure \mathcal{S} over a vocabulary Σ consists of a domain D_T , for every type T and an interpretation for a subset of the other elements of Σ . A theory \mathcal{T} over a vocabulary Σ consists of a set of FO sentences and a set of inductive definitions. Inductive definitions Δ are sets of rules of the form $\forall \bar{x} : P(\bar{x}) \leftarrow \varphi(\bar{y})$, with $\bar{y} \subset \bar{x}$ and $\varphi(\bar{y})$ a FO(\cdot)^{IDP} formula. We call predicate P the defined predicate, or the head of the definition. Any other predicate or function symbol in Δ is called a parameter. The semantics used for these definitions are the well-founded semantics, as argued in [12] this captures the intended meaning of all common forms of definitions and extends the least model semantics of Prolog for negations. Informally a structure \mathcal{S} satisfies Δ if the interpretation of a defined predicate P in the well-founded model of \mathcal{S} ,

constructed relative to the restriction of S to the parameters of Δ is exactly the relation P^S . Another extension in $\text{FO}(\cdot)^{\text{IDP}}$ are the aggregates, these are functions over a set of domain elements, which map such a set to the sum, minimum, maximum, cardinality or product of the elements in that set.

A special property of the $\text{FO}(\cdot)$ family of languages is that they are a “true” declarative logic. They can be used to create a specification of knowledge, not to formulate the description of a problem. Hence, it has no operational semantics and has no unique form of inference. This however does not mean we have no interest in solving problems with these logics. On the contrary, the disengagement of knowledge from problem solving and inferences makes that a specification of domain knowledge may be reused for solving multiple tasks and problems [13]. This leads to the idea of IDP, a knowledge base system, that manages a specification, and provides multiple inferences on it, in order to solve a whole range of problems using a single logic specification.

3.2 The knowledge base system : IDP

We define a knowledge base system (KBS) as a system that supports multiple inferences that can be used on a single specification to be able to execute a range of tasks. The IDP (*Imperative Declarative Programming*) framework is such a KBS, it combines a declarative specification¹, in $\text{FO}(\cdot)^{\text{IDP}}$, a set of inferences and an imperative manipulation of the specification via the Lua [19] imperative programming environment [11]. We will focus only on the most important inferences, for others and more details, we refer to [10].

Modelchecking Given a structure I , a theory T , modelchecking outputs true iff $I \models T$

Model expansion (mx) Given a vocabulary V , a $\text{FO}(\cdot)$ theory T , and a three-valued structure I , that contains a domain D_T for every type in T , model expansion outputs two-valued structures I' such that every $I' \models T$.

Propagation Given a $\text{FO}(\cdot)$ theory T and a three-valued structure I , propagation returns a new three-valued structure I' such that I' approximates every model of T and is more precise than I .

Deduction Given a $\text{FO}(\cdot)$ theory T and a FO sentence φ , deduction outputs true iff $T \models \varphi$. Note: this method is sound but incomplete.

Progression In [8], LTC-theories (Linear Time Calculus) are proposed, a syntactic subclass of $\text{FO}(\cdot)$ theories that allow to naturally model dynamic systems. Given an LTC theory and a structure I_n that provides information about the state of the system on a time point n , the *progression* inference can be used to compute the state (or the possible states) at time point $n+1$ as a new structure I_{n+1} . Repeating the process, we can compute all subsequent states, effectively simulating the dynamic system defined by T . An LTC-theory consists of 3 types of constraints: constraints about the initial situation ($P(0)$), invariants ($\forall t : P(t) \vee Q(t)$), and “bistate” formula’s that relates the state on the current point in time with that of the next ($\forall t : P(t) \Rightarrow P(t+1)$).

¹ We use IDP syntax in the examples throughout the paper. Each IDP operator has an associated logical operator, the main (non-obvious) operators being: $\&(\wedge)$, $\text{---}(\vee)$, $\sim(\neg)$, $!(\forall)$, $?(\exists)$, $<=>(\equiv)$, $\sim=(\neq)$.

The imperative programming environment supports a rich set of operators and inferences to take the logical building blocks in an $\text{FO}(\cdot)^{\text{IDP}}$ specification (vocabularies, theories, terms and structures) and use these to manipulate them and solve more complex reasoning tasks.

The efficiency of the IDP framework in solving problems has been proved in different settings, like for example the most recent ASP-competitions ([14], [9], [1]) and in applications ([7], [21]). Also the different parts of the system have proved their use in multiple situations: the search algorithm MINISAT(ID) has been demonstrated in [2], where it turned out to be the single-best solver in their MiniZinc portfolio, and in the latest MiniZinc challenges [20]. Next to this IDP is used as a didactic tool in various logic-oriented courses, at KU Leuven and at the University of Luxembourg, among others because of its close adherence to first-order logic (FO) and its support for deduction.

We look at a specific small example, the connected graph problem (Listing 1.1), given a graph, we want to know if it is fully connected. In our vocabulary, we have a type `node` (the domain of nodes in the graph), a predicate `edge(node,node)` (there is an edge between these two nodes) and a predicate `reaches(node,node)` (this expresses the reachability relation between two nodes). The theory contains our definitions and constraints. We have one (inductive) definition in this theory, which defines `reaches` (definitions are given between “{” and “}”). Next to this, the theory contains 1 constraint: Every 2 nodes should be reachable from one another.

Besides these 3 logical building blocks, we also have the procedural Lua part. In this code the solver is called to check a model and print out if the graph is fully connected or not.

Listing 1.1. Calling `main()` solves the graph connectivity problem for the given data.

```
vocabulary sp_voc {
  type node
  edge (node , node)
  reaches (node , node)
}
theory sp_theory1: sp_voc {
  { reaches (x,y) <- edge(x,y).
    reaches (x,y) <- ?z: reaches (x,z) & reaches (z,y).    }
  !x y: reaches (x,y).
}
structure sp_struct: sp_voc {
  node = {A..D} // shorthand for A,B,C,D
  edge = {A,B; B,C; C,D; A,D} // ';' separated list of tuples
}
procedure main() {
  sol = modelextend(sp_theory1 , sp_struct , lengthOfPath)[1]
  if (sol == nil) // If no result is returned, no models exist
  then print("The graph is not fully connected.\n")
  else print("The graph is fully connected.\n")
  end
}
```

Note that the concept of inductive definitions is essential to express what reachable means. They capture the construction of the reachability relation. First we take all edges: if there is an edge from node x to node y , then y is reachable from x . When we have added all the edges to *reaches*, we start using these to recursively add new tuples to our relation. If we would try to capture this without definitions, we would soon notice that this is impossible. Say we use the material implication to model the reaches relation:

$$\begin{aligned} reaches(x, y) &\Leftarrow edge(x, y). \\ reaches(x, y) &\Leftarrow \exists z : reaches(x, z) \wedge reaches(z, y). \end{aligned}$$

The relation *edge* can now be empty even though every pair of nodes is in *reaches*. If we use an equivalence, we get following formulation:

$$reaches(x, y) \Leftrightarrow edge(x, y) \vee \exists z : reaches(x, z) \wedge reaches(z, y).$$

Still we can have a model in which *edge* is empty and every possible tuple $(x, y) \in reaches^I$.

4 Modelling the revocation schemes in IDP

Aucher et al. [3] presented a formalization of the eight revocation schemes introduced in section 2 in a dynamic variant of propositional logic that resembles imperative programming languages. In this section we sketch our implementation² of the eight revocation schemes and the undoing operation in IDP. Because of the nature of IDP, whose inductive definitions suit the recursive character of the revocation schemes very well, the revocation schemes could be implemented in a very straightforward way. The implementation sheds light on both the formal properties and the practical implications of the revocation schemes, and can thus support a developer of an access control system in her decisions concerning the precise nature of the revocation schemes to be included in the system.

Unlike the formal definition in [3], our implementation does not work by implementing each of the eight revocation schemes separately, but by specifying the formal properties of the three dimensions of the classification in an IDP theory. IDP can then execute the revocation schemes based on this formal specification.

In the sketch of the IDP implementation, we concentrate on the four deletion schemes. The only additional complication in the four negative schemes is the labelling system for keeping track of what to do in the undoing operation.³

4.1 Preliminaries: Vocabulary and auxiliary predicates

The IDP theory models the change of the authorization specification over time. Principals are modelled as objects of the theory's domain, whereas authorizations are modelled by a partial function (for positive authorizations) and a predicate (for negative authorizations) on pairs of principals. The authorizations cannot be modelled as objects, because they change over time, while IDP assumes a constant domain of objects.

² The implementation can be downloaded at <http://icr.uni.lu/mcramer/index.php?id=3>.

³ This labelling system is well-documented in the comments to the code of our implementation.

The partial function `pos_auth` that models positive authorization can take `TT` and `TF` as values, depending on whether it represents an authorization of the form (i, j, \top, \top) or (i, j, \top, \perp) . Apart from the two principals i and j , it takes a point in time as argument. Negative authorizations are modelled by a separate predicate called `FF`, also taking a point in time and two principals as arguments. The reason for this separation of positive and negative authorizations is that it does not make sense to have two different positive authorizations linking the same pair of principals, whereas it does make sense to have a negative authorization additionally to a positive authorization linking the same pair of principals.

The objects `TT` and `TF` that serve as values of `pos_auth` are given a separate type called `authorization`. The other types of objects in the domain of the IDP theory are points in time, principals and revocation schemes.

Listing 1.2. The vocabulary of the IDP implementation

```
vocabulary V{
  type time isa int // Points in times are integers
  type principal
  type scheme
  type authorization

  SOA: principal
  TT: authorization
  TF: authorization
  WGD: scheme // Weak Global Delete
  WLD: scheme // Weak Local Delete
  SGD: scheme // Strong Global Delete
  SLD: scheme // Strong Local Delete
  WGN: scheme // Weak Global Negative
  WLN: scheme // Weak Local Negative
  SGN: scheme // Strong Global Negative
  SLN: scheme // Strong Local Negative
  UN: scheme // Undo operation

  // Positive and negative authorizations
  partial pos_auth(time, principal, principal): permission
  FF(time, principal, principal)
  // Start configuration of authorizations
  partial pos_auth_start(principal, principal): permission
  FF_start(principal, principal)

  // Auxiliary predicates
  active_chain(time, principal)
  ind(time, principal, principal)
  access_right(time, principal)

  // Predicate for specifying which revocation schemes to apply
  rs(time, scheme, principal, principal)

  // Changes on the authorization specification
```

```

delete(time, principal, principal)
partial new(time, principal, principal): permission
}

```

`pos_auth` is defined inductively by setting its values at time $t = 0$ to the start configuration specified by `pos_auth_start`, and by modifying its values between time t and time $t + 1$ according to the changes specified by `delete` and `new`:

Listing 1.3. The definition of the authorization specification at a given time

```

{ pos_auth(0, p1, p2) = x <- pos_auth_start(p1, p2) = x.
  pos_auth(t+1, p1, p2) = x <- pos_auth(t, p1, p2) = x & ~delete(t, p1, p2).
  pos_auth(t+1, p1, p2) = x <- new(t, p1, p2) = x. }

```

Since in this sketch of the implementation we are leaving out the negative revocation schemes, we can ignore negative authorizations. In the actual implementation, there are predicates `FF_delete` and `new_FF` that specify changes on the negative authorizations, and `FF` is defined in a way analogous to `pos_auth` using these change predicated instead of `delete` and `new`.

The auxiliary predicates `active_chain`, `ind` and `access_right` model the existence of an active rooted delegation chain for a principal, the independence of a principal from another principal and the access right of a principal. Their definitions in IDP correspond directly to the definitions of the corresponding notions in section 2:

Listing 1.4. The definitions of the auxiliary predicates

```

{ active_chain(t, SOA).
  active_chain(t, p1) <- ?p2: active_chain(t, p2) & pos_auth(t, p2, p1) = TT & ~FF(t, p2, p1). }
{ ind(t, SOA, p).
  ind(t, p1, p2) <- ?p: ~p = p2 & ind(t, p, p2) & pos_auth(t, p, p1) = TT & ~FF(t, p, p1). }
{ access_right(t, p) <- active_chain(t, p).
  access_right(t, p) <- ?p1: active_chain(t, p1) & pos_auth(t, p1, p) = TF & ~FF(t, p1, p). }

```

4.2 Specifying propagation and dominance for deletion schemes in the IDP theory

The meaning of *local* vs. *global* propagation is captured by the inductive definition of the partial function `new`, which specifies which new authorizations are added to the authorization specification:

Listing 1.5. The definition of `new` captures the propagation dimension

```

{ new(t, i, k) = x <- ?j s: (s = WLD | s = SLD | s = WLN | s = SLN) & rs(t, s, i, j) & ~active_chain(t+1, j) & pos_auth(t, j, k) = x. }

```

Informally, this definition says that if in a local revocation scheme revoking a positive authorization from principal i to principal j , j is losing its delegation right, then every

positive authorization from j to another principal k must be replaced by a positive authorization of the same authorization type from i to k .

In order to understand why this definition of `new` also ensures that the propagation of the deletion is blocked in local revocation schemes in the desired way, we need to look at the definition of `delete`. It is defined using an inductive definition with four clauses:

Listing 1.6. The definition of `delete` captures the dominance dimension

```
{ delete(t, i, j) <- rs(t, s, i, j) & (s=WLD | s=WGD | s=SLD | s=SGD) .
  delete(t, i, j) <- pos_auth(t, i, j)=x & ~active_chain(t+1, i) .
  delete(t, k, j) <- rs(t, SLD, i, j) & pos_auth(t, k, j)=x & ~ind(t, k, i) .
  delete(t, z, w) <- rs(t, SGD, i, j) & delete(t, p, w) & pos_auth(t, z, w)=
    x & ~ind(t, z, i) .
```

Let us first concentrate on the first two clauses: The first clause just states that in any deletion revocation scheme from i to j , the positive authorization from i to j is deleted. The second clause defines the propagation of deletion by specifying that any positive authorization from i to j gets deleted if i is losing its delegation right. Since in local revocation schemes, the definition of `new` ensures that principals who had previously received their delegation right from j will now receive it from i , the propagation gets blocked after j in local revocation schemes, as desired.

The meaning of *strong* vs. *weak* dominance is captured by the third and fourth line of the inductive definition of `delete`: These lines specify the additional deletions that are needed in strong revocation schemes.

Note that we needed to specify the additional strength of the deletion separately for strong local deletes and strong global deletes: This is because we wanted – in line with the definition of the strong global delete in [18] and [3] – a strong global delete from i to j not only to be strong in the sense of deleting other permissions to j dependent on i , but also to delete other permissions dependent on i to descendants of j . We doubt, however, whether this additional strength of the strong global delete would actually be desirable in a real access control system: Strong revocation schemes are usually applied to distrusted principals, whose rights one wants to restrict as much as possible. But there is no reason why another principal, who has a rooted delegation chain independent of this distrusted principal, should have his rights removed only because he also has a rooted delegation chain dependent on the distrusted principal. The version of the strong global delete that we judge more reasonable is the one in which the fourth line of the inductive definition of `delete` is removed and the third line is also applied to the strong global delete.

This discussion of the details of the strength of the strong global delete illustrates how modelling revocation schemes in IDP can shed light on the properties of the revocation schemes in a way that can support a developer of an access control system in fixing the specification of the schemes to be implemented in the system.

5 Related Work

The classification of revocation schemes used in this paper was first introduced by Hagström et al. [18]. Their paper, however, was rather informal in nature.

The first formalization of this classification was presented by Aucher et al. [3]. They use a dynamic variant of propositional logic for their formalization. Unlike our specification of the revocation schemes in IDP, their formalization required all eight revocation schemes to be formalized separately.

Barker et al. [4] have represented delegation-revocation models in terms of reactive Kripke models [16]. They implement this approach by translating first-order representations of the reactive Kripke models into an equivalent Answer Set Programming form. Answer Set Programming is a logic programming approach close in nature to IDP.

The IDP system is maturing, as is shown by applications in multiple fields of interest. In [7] a set of machine learning applications have been solved by an approach with the IDP system and the FO(\cdot) framework. Among others, this research showed a very elegant and efficient solution for the stemmatology application. Given different versions of an ancient text, the goal of stemmatology is to find which one is the original and which text is copied from which. The problem was specified in a theory of one sentence and was able to solve large instances.

Another application in which IDP was put to the test was in [21], in which a typical application from Business Rule Systems was taken and the behaviour was modelled in the IDP system. A comparison between the IDP and the Business Rule approach was made. The IDP system had some great advantages, like the possibility to reason in context of incomplete knowledge, the ability to reason hypothetically and in multiple directions.

6 Conclusion

We have shown, how the knowledge base system IDP can be used for efficiently implementing the revocation schemes in Hagström et al.'s [18] classification. This implementation works by specifying the properties of the three dimensions of the classification in an IDP theory. By using the model expansion inference of IDP, this declarative specification becomes an executable program implementing the eight revocation scheme in the classification. We also illustrated how the IDP implementation can help to shed light on the formal properties of the revocation schemes and can thus support the development of ownership-based access control systems.

References

1. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The Fourth Answer Set Programming Competition: Preliminary Report. In: Cabalar, P., Son, T.C. (eds.) LPNMR. Lecture Notes in Computer Science, vol. 8148, pp. 42–53. Springer (2013)
2. Amadini, R., Gabbrielli, M., Mauro, J.: Features for Building CSP Portfolio Solvers. arXiv:1308.0227 [cs.AI] (2013)
3. Aucher, G., Barker, S., Boella, G., Genovese, V., van der Torre, L.: Dynamics in delegation and revocation schemes: A logical approach. In: Li, Y. (ed.) Data and Applications Security and Privacy XXV, Lecture Notes in Computer Science, vol. 6818, pp. 90–105. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-22348-8_9

4. Barker, S., Boella, G., Gabbay, D.M., Genovese, V.: Reasoning about delegation and revocation schemes in answer set programming. *Journal of Logic and Computation* 24(1), 89–116 (2012)
5. Bertino, E., Samarati, P., Jajodia, S.: An extended authorization model for relational databases. *Knowledge and Data Engineering, IEEE Transactions on* 9(1), 85–101 (Jan 1997)
6. Bertino, E., Jajodia, S., Samarati, P.: A non-timestamped authorization model for data management systems. In: *Proceedings of the 3rd ACM Conference on Computer and Communications Security*. pp. 169–178. CCS '96, ACM, New York, NY, USA (1996), <http://doi.acm.org/10.1145/238168.238211>
7. Blockeel, H., Bruynooghe, M., Bart, B., De Cat, B., De Pooter, S., Denecker, M., Labarre, A., Ramon, J., Verwer, S.: Predicate Logic as a Modeling Language: Modeling and Solving some Machine Learning and Data Mining Problems with IDP3. CoRR abs/1309.6883 (2013)
8. Bogaerts, B., Jansen, J., Bruynooghe, M., Cat, B.D., Vennekens, J., Denecker, M.: Simulating Dynamic Systems Using Linear TimCalculus Theories. In: *Submitted to ICLP '14* (2014)
9. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming System Competition: Preliminary Report of the System Competition Track pp. 388–403 (2011)
10. De Cat, B., Bogaerts, B., Bruynooghe, M., Denecker, M.: Predicate Logic as a Modelling Language: The IDP System. CoRR abs/1401.6312 (2014)
11. De Pooter, S., Wittocx, J., Denecker, M.: A prototype of a knowledge-based programming environment. In: *International Conference on Applications of Declarative Programming and Knowledge Management* (2011)
12. Denecker, M., Ternovska, E.: A Logic of Nonmonotone Inductive Definitions. *ACM Transactions on Computational Logic (TOCL)* 9(2), 14:1–14:52 (Apr 2008)
13. Denecker, M., Vennekens, J.: Building a Knowledge Base System for an Integration of Logic Programming and Classical Logic. In: García de la Banda, M., Pontelli, E. (eds.) *ICLP. LNCS*, vol. 5366, pp. 71–76. Springer (2008)
14. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The Second Answer Set Programming Competition. In: *LPNMR*. pp. 637–654 (2009)
15. Fagin, R.: On an authorization mechanism. *ACM Trans. Database Syst.* 3(3), 310–319 (Sep 1978), <http://doi.acm.org/10.1145/320263.320288>
16. Gabbay, D.M., Marcelino, S.: Modal logics of reactive frames. *Studia Logica* 93, 405–446 (2009)
17. Griffiths, P.P., Wade, B.W.: An authorization mechanism for a relational database system. *ACM Trans. Database Syst.* 1(3), 242–255 (Sep 1976), <http://doi.acm.org/10.1145/320473.320482>
18. Hagström, Å., Jajodia, S., Parisi-Presicce, F., Wijesekera, D.: Revocations-a classification. In: *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*. pp. 44–. CSFW '01, IEEE Computer Society, Washington, DC, USA (2001), <http://dl.acm.org/citation.cfm?id=872752.873508>
19. Ierusalimsky, R., de Figueiredo, L.H., Celes, W.: Lua – An Extensible Extension Language. *Software: Practice and Experience* 26(6), 635–652 (1996), [http://dx.doi.org/10.1002/\(SICI\)1097-024X\(199606\)26:6<635::AID-SPE26>3.0.CO](http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO)
20. Minizinc challenge 2012. <http://www.minizinc.org/challenge2012/results2012.html>
21. Van Hertum, P., Vennekens, J., Bogaerts, B., Devriendt, J., Denecker, M.: The effects of buying a new car: an extension of the IDP Knowledge Base System. *TPLP* 13(4-5-Online-Supplement) (2013)